Design and Evaluation of Decentralized Scaling Mechanisms for Stream Processing

Mehdi Belkhiria, Cédric Tedeschi

Design and Evaluation of Decentralized Scaling Mechanisms for Stream Processing

Mehdi Mokhtar Belkhiria
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
mehdi.belkhiria@irisa.fr

Cédric Tedeschi
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
cedric.tedeschi@inria.fr

***Abstract*—Over the last decade, stream processing (SP) gained momentum as an efficient paradigm dealing with the real-time processing of large streams of data. As mature as they are, mainstream stream processing engines are not ready to be used over more distributed platforms that are today emerging such as Fog platforms. Streams being produced at the edge of the platform, and edges being composed of less powerful compute nodes, a decentralized approach of Stream Processing is needed. This paper moves a step towards decentralized SP by proposing a decentralized autoscaling mechanism for SP applications. Autoscaling refers to the ability for a system to automatically scale in and out as the load varies. As a stream processing application is generally composed of a set of operators in a pipeline, our protocol lets each operator's replica take scaling decisions and enforce them independently. Assuming a fully decentralized setting, we consider each replica has only a partial view of the graph of operators. Then, without care, concurrent distributed scaling could lead to inconsistencies in replicas' views and to data loss. To address this issue, we devise a maintenance protocol ensuring the consistency of replicas' views, no data loss and no downtime during reconfigurations. Finally, the paper describes the early experimentations conducted with a software prototype of a decentralized SP engine over a computing cluster, showing the usability of the mechanism.**

I. Introduction

Stream Processing has been developed as an answer to the need for real time processing of continuous, large streams of data. Stream Processing is becoming a ubiquitous paradigm in various areas, ranging from smart cities to social media. Stream Processing typically implements data processing pipelines. Each data item goes through a set of operators to be applied on the data. Two data items can be processed at the same time but at different stages of the workflow. More generally speaking, a stream processing application can be represented by a directed acyclic graph (DAG) where nodes are operations to be applied on each data item, and edges represent the streams of data between operators.

Putting Stream Processing into practice, Stream Processing Engines (SPEs) ease the development and deployment of Stream Processing Applications, through two main features, namely, i) a high-level API to define the operators and their dependencies, and ii) an automated deployment of the application over a distributed computing platform. Storm [12], Flink [6] and Spark Streaming [23] are examples of these SPEs which are today regarded as mature Stream Processing frameworks that can be used in multiple contexts typically integrating IoTs.

Incoming streams of data can vary a lot over different timescales, and sudden bursts or reductions in the velocity of the streams are to be expected in the lifetime of an application. This variability calls for automated mechanisms able to scale the resources supporting the operators dynamically to adjust the allocation to the needs. Such an elastic autoscaling allows to ingest bursts gracefully while avoiding computing power wastage and its associated cost.

With the advent of more geographically dispersed computing platforms as described in Edge or Fog computing paradigms [22] which typically targets IoT-based applications where data streams are to be handled as close as possible to the user, Stream Processing Engines need to support a decentralized deployment of applications. Over this new kind of platforms, operators will typically get spread over the platform. This decentralization also applies to management as keeping a consistent view of the global platform in the absence of a centralized stable infrastructure, taking decisions and enforcing them becomes intractable.

This paper adopts a vision of a decentralized management for SPEs and applies it to the autoscaling problem. We consider a model where each operator — and even each replica of an operator in the DAG becomes responsible for its own scaling. The operator has only a local view of the system, and takes scaling decisions independently, according to its own experience of the load. As replicas are replicating themselves, there is a need to maintain the local views of operators neighbouring the scaled operator so as no record is lost in the data stream. The contributions made in this paper are the following: (i) A decentralized scaling mechanism within which decision to scale in or out is taken by each replica independently, (ii), a maintenance protocol to keep views updated as replicas appear and disappear, as decided by others without any downtime in the process, and (iii) an early software prototype of a decentralized Stream Processing Engine, that was deployed over a computing cluster which is part of the Grid'5000 [3] computing platform.

Recent efforts put into autoscaling are shown in Section II. The decentralized algorithm and its associated maintenance protocol is presented in Section III. Results from simulation experiments are provided in Section IV. The design and implementation of a Stream Processing Engine is discussed in Section V, as well as early experiments conducted with it

over the Grid'5000 platform.

II. Related Work

The need for autoscaling in Stream Processing has two main sources. Firstly, the incoming stream is subject to changes in its velocity which is hard to predict. Secondly, each operator has its own latency, also hard to predict and differs between operators. Autoscaling is a topic which recently gained attention from the research community [10], [15]. Approaches for autoscaling in SP systems can be classified along different dimensions. Yet, we generally distinguish static from dynamic approaches.

a) Static Scaling: A static approach to scaling generally consists of a prior-to-execution analysis of the graph of operators in order to *discover* what portions of the graph can be parallelized safely. This can be done using heuristics traversing the graph and marking contiguous stateless operators, which can be generally safely duplicated [19]. Stateful portions of the graph are more difficult to make parallel as some state needs to be maintained in a distributed fashion. Such a static analysis is generally envisioned as a necessary first step towards scaling. It does not dynamically adjust the number of replicas of an operator as the input stream rate evolves.

b) Building blocks for Dynamic Scaling: Dynamic scaling generally relies

on three operations: fusion, fission, and deletion [15]. *Fusing* two contiguous operators that are hosted over two different hosts means that they start running on the same compute node. This allows to reduce traffic between operators at the cost of a possible less perfect load balance. Fusion is not a scaling action *per se*, and relates more to a consolidation of the placement of operators over the compute nodes. *Fission* refers to an operator's duplication. It scales it out by spawning a new *replica* (or *instance*[1]) of the operator, thus increasing its level of parallelism. This increase is effective provided the new thread or process started leverages either previously unused CPU power of an already allocated machine (vertical scaling), or that of a node allocated for this purpose (horizontal scaling [17], [18]). Scaling out a stateful operator brings few difficulties: If partitioned, the state of an operator is split over its different replicas. When a new instance appears, part of this partition needs to be migrated to the new node. Conversely, when some replica disappears, its partial state needs to be dispatched over remaining nodes. If not partitionable, a stateful operator can be scaled provided a way to merge partial states computed concurrently by the replicas. Statefulness is not our primary concern here but existing approaches could be used to extend our scaling algorithm to stateful operators [9]. *Deletion* is fission's inverse operation. It consists of removing running instances of a given operator, typically when the operator's incoming load gets reduced. This is the building block for a scale-in operation.

[1]*Replica* and *instance* are used interchangeably hereafter.

c) Dynamic Scaling in Practice: To be put into action, dynamic scaling needs two elements [9], [13], [14], [21]. Firstly, up-to-date information about the load of nodes and the available resources has to be collected, to be able to take relevant decisions. Secondly, a scaling policy to decide when a scaling operation is needed. Some works monitor the CPU utilization, detect bottlenecks and trigger a scaling-out phase, in particular for partitioned stateful operators, which requires to split and migrate the state of the operator between the evolving set of instances [9]. T-Storm [21] introduces a mechanism of dynamic load rebalance triggered periodically into Storm, with a focus on trying to reduce internode communication by grouping operators. Aniello et al. followed the same approach similarly [2]. StreamCloud [14] describes a set of techniques to identify parallelizable zones in a pipeline of operators. The pipeline is split into zones that starts and ends by stateful operators in a way which is similar to what can be found in some static approaches [19]. After splitting, dynamic scheduling is introduced so as to balance the load at the entry point of each zone. Another work tries to maintain the SASO properties (Settling time, Accuracy, Stability and Overshoot) in a Stream Processing applications through a combination of scale-out and scale-in operations [13]. More precisely, it attempts at dynamically allocating the right amount of instances ensuring the performance of the system (accuracy), to do it quickly (settling time), that it does not oscillate artificially (stability) and that no resource used for nothing (no overshoot). While their objectives are similar to those of the present work, all these works, in contrast to the work presented in the following, rely over a centralized authority to monitor the system, decide on the scaling operations and enforce them.

d) Towards Decentralized Stream Processing Management.: Decentralizing the management of stream processing has been the subject of few works [5], [7], [8], [16], [18]. DEPAS [5] does not specifically target Stream Processing. It decentralizes scheduling decisions in a multi-cloud infrastructure over local schedulers. The similarity between DEPAS and the present work stands in that a set of autonomous instances take probabilistic scaling decisions to reach a global desirable result. More specifically targeting stream processing, Pietzuch et al. [18] proposed a Stream-Based Overlay Network (SBON) that allows to distribute stream processing operators over the physical network. Hochreinter et al. [16] devise an architectural model to deploy distributed stream processing applications. Finally, Cardellini et al. [7], [8] partially decentralize autoscaling in SP through a hierarchical approach combining a threshold-based local scaling decision with a central coordination mechanism to decide what decisions will actually get enforced. Autoscaling generally leads to a pause-and-restart phase: when a scaling operation takes place, the application is paused. It gets restarted once reconfiguration is done. As mentioned before, this reconfiguration is more complex when impacting partitioned stateful operators. In the following, we devise a fully-decentralized autoscaling protocol that does not require to pause data processing during reconfigurations. To make the problem simpler in a first step,

we assumed stateless operators.

To our knowledge, no such fully-decentralized proper protocol was proposed assuming neither stateful, nor stateless protocols. A preliminary version of the algorithms and concepts presented in the following has been described in an early stage in [4]. The aforementioned paper contains an early version of the protocol and focus on the analysis of its correctness. The present paper presents the detailed algorithms, captures the potential efficiency of such an approach through simulations and complexity analysis, and presents the early prototyping and experimentation of a decentralized stream processing engine.

III. Decentralized Graph Autoscaling

This section presents the core component of our decentralized Stream Processing Engine: its scaling mechanism. As a first step, we assume stateless operators and a relatively optimistic system model, defined below. The algorithm allowing each node to decide independently to scale or not and up to what level, as well as the associated graph maintenance protocol is then described in Sections III-B and III-C.

A. Model

e) System Model: We consider a distributed system composed of an unbounded set of (geographically dispersed) homogeneous compute nodes. These nodes can be either physical or virtual machines. We assume that allocating a new compute node is abstracted out through the `createNode()` primitive. While *homogeneity* could seem an unrealistic assumption at first, in practice it simply means that all virtual machines allocated must have the same size, which is a simple property to achieve in many Cloud contexts. Also, these nodes are assumed to be reliable: we consider only *fair leaves*: any departing event is known and handled gracefully before the departure is actually committed. In other words, nodes cannot crash or leave without notice. Nodes communicate in a partially synchronous model [11] using FIFO reliable channels: A message reaches its destination in a finite time, and two messages sent through the same channel are processed in the same order they were sent. Sending a message is done using the `send(type,ctnt,dest)` non-blocking method. `type` denotes the message type and `ctnt` its content. The actual structure of `ctnt` varies depenting on the value of `type`. `dest` is the address of the destination node. The higher-level communication primitive `sendAll(type, ctnt, dests)` sends the same message to all nodes in `dests`.

f) Application Model: We consider stream processing applications represented as directed pipelines in which vertices represent operators to be applied on each input record and edges represent streams between these operators. We assume stateless operators. At starting time, each operator is launched on one particular compute nodes, and each compute node hosts a single replica. Then, the scaling mechanism can add or remove replicas. Each replica of an operator is referred to as an *operator instance* (OI) in the following. OIs running the same operator are referred to as *siblings*. The load of an operator is shared equally between all of its instances. Each operator O_i can exist in several instances OI_{ij} where i is the id of the operator and j the id of the instance. In the example of Fig. 1, the pipeline is composed of five operators. At some point, scaling out introduced two new instances for the middle operator. The application follows a purely distributed configuration: due to the geographic dispersion of nodes and for the sake of scalability, the view of the graph on each instance is limited to the instances of their successor and predecessor operators. For example, in this configuration, the neighbour table of instances of operator $O3$ have only three entries: the address of the single instance of Operator $O2$ and the addresses of the two instances of Operator $O4$. It is assumed that the incoming load of an operator is evenly shared amongst its instances.

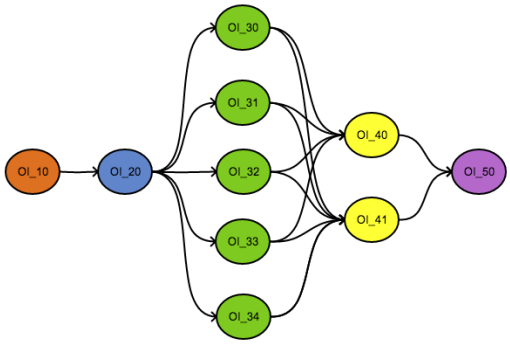

Fig. 1. A scaled 5-stage pipeline.

Periodically, each instance triggers the decision phase. During this phase, the current load is checked to decide whether some scaling action is needed. It is assumed that the incoming load of an operator is evenly shared amongst its instances, so instances are able to take uncoordinated decisions leading to a global accurate number of instances to handle the load, as described in Section III-B. Once an OI decides to get duplicated or deleted, it actually executes the action planned and ensures its neighbours are informed of it through the maintenance protocol described in Section III-C. Note finally, that there is a one-to-one mapping between allocated compute nodes and an operators' instances.

B. Decentralized Scaling Policy

The scaling mechanism gets triggered periodically on each node. Its pseudo-code is shown in Algorithm 1: Two cases trigger a scaling action: when the load is either above an upper threshold, or below a lower threshold.

Algorithm 1 Periodic autoscaling mechanism.

```
1: if l_t ≥ thres_↑ then
2:     operatorScale − Out()
3: else if l_t ≤ thres_↓ and !isLeader then
4:     operatorScale − In()
5: end if
```

Let us consider one node, *i.e.*, one instance OI of an operator O. Let C denote the capacity of nodes, *i.e.*, the number of records they can process per time unit. Let l_{curr} the current load experienced by the instance, *i.e*, the number of records received during the last time unit. Finally, r, with $0 < r \leq 1$ denotes the desired load level of operators, typically a parameter set by the user. It represents the targeted ratio between load and capacity of each node. The objective for an instance is to find the replication factor to be applied to itself so all instances of this operator reach a load level of r. Each node contributes to the targeted scale equally by inferring a *local replication factor*. The desired load for an OI is $r \times C$, which means that this OI needs to be scaled with a factor of $|\frac{l_t}{r \times C}|$. Note that this factor will be concurrently calculated and applied by each OI for this operator. This means that the OI will need to get duplicated $p = |\frac{l_t}{r \times C} - 1|$ times, which expresses the ratio between the local target and the local current situation is computed. p is interpreted differently according to the two possible cases: (i) If we were in the case of a load exceeding the upper threshold, then, p is a replication factor. If $p < 1$, p is interpreted as a duplication probability: the node will get duplicated with probability p. Otherwise, the node will get duplicated $\lfloor p \rfloor$ times and then one final time with probability $p - \lfloor p \rfloor$. (ii) If we were in the case of a load being below the lower threshold, p is a termination probability. Note that there is a risk that all instances of O take this decision at the approximate same time, leading to a collective termination, and to O's disappearance . This problem is solved by introducing a particular node (called the *operator keeper*) that cannot terminates itself whatever its load is. Yet, such a probabilistic distributed decision is subject to the possibility of taking a *bad* decision, especially when there are only a small number of instances. When the number of instances increases, the probability of starting or terminating a non-accurate global number of instances drops rapidly. In the following pseudo-code, this procedure is materialized by a call to the `getProbability()` function, and the `applyProba(p: real)` function that transforms a probability into a boolean stating whether the deletion or duplication action will actually take place.

C. Scaling and Maintenance Protocol

Let us now explain the global protocol, including the maintenance of the graph which allows to avoid downtime while scaling operations are performed concurrently. Algorithm 2 gives the pseudo-code of the protocol triggered once an instance decided to start a scale-out operation. The first part of the algorithm consists in calculating the amount of duplication needed to reach the targeted load ratio r (in Lines 2-4, and as described in the previous section). From Lines 5 to Lines 7, the calculated amount of nodes gets started. These newly spawned instances are not yet `active`: they are idle, waiting for a specific message of the current instance to initialize their neighbours table and start processing incoming data (which can be stored in some queue in the meantime). During that time, the current node, in Lines 9-11, spreads the information of these new nodes to its own neighbours and waits for their acknowledgement. As scaling operations can be performed concurrently, this confirmation protocol ensures no neighbours left in the meantime. This aspect of the protocol is more formally discussed in [4].

Lines 14-25 gives the protocol on the predecessors' or successor's side, on which the addresses of the new nodes are added in the corresponding table. If the node receiving the message is itself not yet `active`, i.e., it is itself a newly spawned node of the neighbouring operator, it stores the new instances in a particular $succsToAdd$ table containing future neighbours. The neighbour acknowledges the message to the duplicating node by sending a $duplication\_ack$ message. Once all acknowledgements have been received by the duplicating instance, the duplicating instance sends a `start` message to its new siblings to activate them (in Line 31). Upon receipt — refer to Lines 35-36) — the new siblings initialize the sets of their neighbors by combining the sets sent by the duplicating instance and the possible information received in the meantime, stored in $*ToAdd$ and $*ToDelete$ variables.

Algorithm 2 Scale-out protocol.

Input: C: nodes' capacity, r: desired load ratio
Input: l_{curr}: current load
Input: $thres_{\uparrow}$: threshold
Input: $succs, preds$: arrays of successors and predecessors

1: **procedure** $opScaleOut()$
2: $p \leftarrow getProbability(C, r, l_{curr})$
3: $newAddrs \leftarrow []$
4: $n \leftarrow \lfloor p \rfloor + applyProba(p)$
5: **if** $n > 1$ **then**
6: **for** $i \leftarrow 1$ **to** n **do**
7: $newAddrs.add(createNode())$
8: **end for**
9: $sendInformation("duplication", succs, preds, newAddrs)$
10: $nbAck \leftarrow 0$
11: $nbAckExpected \leftarrow |succs| + |preds|$
12: **end if**
13: **upon** receipt **of** $("duplication", addrs)$ **from** p
14: **if** $p \in$ succs **then**
15: **if** isActive **then**
16: $succs = succs \cup addrs$
17: **else**
18: $succsToAdd = succsToAdd \cup addrs$
19: **end if**
20: **else if** $p \in$ preds
21: **if** isActive **then**
22: $preds = preds \cup addrs$
23: **else**
24: $predsToAdd = predsToAdd \cup addrs$
25: **end if**
26: $send("duplication\_ack", p)$
27: **upon** receipt **of** $("duplication\_ack")$
28: $nbAck++$
29: **if** $nbAck = nbAckExpected$ **then**
30: **for all** $newSibling$ **in** $newAddrs$ **do**
31: $send("start", succs, preds, newSibling)$
32: **end for**
33: **end if**
34: **upon** receipt **of** $("start", succs\_, preds\_)$ **from** p
35: $succs = succs\_ \cup succsToAdd \setminus succsToDelete$
36: $preds = preds\_ \cup predsToAdd \setminus predsToDelete$
37: $isActive \leftarrow true$

The scale-in protocol, detailed in Algorithm 3, triggered once a node decides to terminate itself according to the policy described above, is simpler than the scale-out protocol. Algorithm 3 shows how the current instance, before self-termination, ensures that every node pertained by the deletion

(its neighbours) is informed of it. On receipt of this upcoming termination notification, we again have to consider two cases, depending whether the receiving node is active or not: if it is, then the node is simply removed from the list of its neighbors (either from $pred$ or $succ$) and an acknowledgement is sent back. Otherwise, the node is stored in a *to-be-deleted* table, that will be taken into account at starting time. The final step consists in waiting for all the acknowledgements of its neighbours to be sure they are informed of it. After that, it flushes its data queue and triggers its own termination.

Algorithm 3 Scale-in protocol.

Input: $thres_{\downarrow}$: threshold
1: **procedure** $operatorScale - In()$
2: $p \leftarrow getProbability(C, r, l_{curr})$
3: **if** $applyProba(p)$ **then**
4: $sendInformation("deletion", succs, preds, me)$
5: $nbAck \leftarrow 0$
6: $nbAckExpected \leftarrow |succs| + |preds|$
7: **end if**
8: **upon** receipt **of** $("deletion", addr)$ **from** p
9: **if** $P \in$ succs **then**
10: **if** isActive **then**
11: $succs \leftarrow succs \setminus addr$
12: **else**
13: $succsToDelete \leftarrow succsToDelete \cup addr$
14: **end if**
15: **else if** $p \in$ preds
16: **if** isActive **then**
17: $preds \leftarrow preds \setminus addr$
18: **else**
19: $predsToDelete \leftarrow predsToDelete \cup addr$
20: **end if**
21: $send("deletion\_ack", p)$
22: **upon** receipt **of** $("deletion\_ack")$
23: $nbAck++$
24: **if** $nbAck = nbAckExpected$ **then**
25: // wait current tuples to be processed
26: $terminate()$
27: **end if**

D. Reducing the Risk of Delayed Records

Relying on probabilistic policies to scale might sometimes lead to decisions which globally result in an approximate level of parallelism. In particular, when some operator is not scaled enough, the incoming load may exceed the capacity, in which case some data will get delayed, or even lost. One first parameter to use to mitigate this risk, is to lower r: the lower r, the higher the calculated number of nodes needed to handle the load. But r being set by the user, the system cannot act on it. Yet offering an extra guarantee to the user whatever r can be done in the following way: Let us remark that to perfectly handle all the messages without incurring any delay, we need $n_{th} = L_{curr}/C$ instances, where L_{curr} denotes the current global load on the operator considered. This means that each node, to reach this ideal, needs to get replicated $p_{th} = \frac{n_{th} - n_{curr}}{n_{curr}}$ times. Yet, by security, and to avoid any probabilistic effect, each node can take $p_{secure} = ceil(p_{th})$ as replication factor. Doing so reduces the risk of undershoot and consequently the risk of delaying message processing. Adopting p_{secure} as the replication factor leads to a number of nodes $n_{secure} = n \times (p_{secure} + 1)$, each node having a load $l_{secure} = L_{curr}/n_{secure}$, and a load ratio which is $r_{secure} = l_{secure}/C$. r_{secure} can be seen as the ratio which will be obtained if we ensure deterministically that the load does not exceed the capacity. Then there are two cases: either $r \leq r_{secure}$ and r can be used safely, or $r > r_{secure}$ and switching r for r_{secure} as the value for the desired load when triggering the policy will bring an extra guarantee while not violating the constraint on r given by the user. Allowing the algorithm to switch to $r = r_{secure}$ in the second case has been included in the algorithm, so as to have an extra guarantee to avoid deleted messages. It is evaluated at the end of the next section.

IV. Simulations

In this section, we evaluate our algorithm, reagarding accuracy, rapidity of scaling, and the extra guarantee regarding delayed messages. We developed a discrete-time simulator in Java. Each time step t sees the following operations: a subset of the nodes tests the conditions for triggering a scaling. In case the protocol is initiated, the first message (`duplication` or `deletion`) is received by the neighbours of the initiating node. Then, messages sent at step t are processed at step $t + 1$ and new resulting messages are sent as per the protocol, to be processed at time $t + 2$, and so on. A scale-out operation spans three steps, and a scale-in one spans two. The variation of the workload is modelled by a stochastic process, mimicking a Brownian motion, which allows us to evaluate our algorithm with significant variations of the workload. The graph tested is a pipeline composed of 5 operators, each operator having a workload evolving independently. Initially, each operator is duplicated on 14 OIs. Compute nodes have a processing capacity of 500 tuples per time step. The other parameters are: $r = 0.7$, $thres_{\downarrow} = 0.6$, and $thres_{\uparrow} = 0.8$. (These parameters were chosen empirically, further investigation would be needed to set them optimally.) Nodes trigger the scaling algorithm every 5 time steps. Yet each OI may start the protocol at different steps.

We start evaluating our algorithm's ability to quickly adapt load's variation and reach an adequate number of instances to maximizing the throughput. For the sake of comparison, we show how a per-operator centralized approach where a single leader replica takes all scaling decisions alone would perform. The following results are given for one operator, and the leader-based approach is referred to as the *centralized* one even if it is not fully centralized, but *per-operator* centralized.

Fig. 2(a) plots the number of OIs with the decentralized approach (blue curve) compared to the number of OIs with the centralized approach (green curve) and the ideal number of OIs (orange curve) which is obtained by dividing the load by the capacity of nodes, the whole multiplied by the ideal load ratio r. We observe that the number of nodes of both approaches scales quickly: The delay between a load variation and the adaptation can be quite reduced. This also shows that nodes are able, without coordination, and only based on decisions using local information, to start or remove nodes in a batch fashion. It means that if X more nodes are needed, X nodes will be added over a short period of time, the burden of starting these X nodes being shared by the existing nodes. To

compare more deeply our approach with the centralized one, we present two other measurements. The first one Fig. 2(b)(c) is the percentage of maximum throughput. A percentage of 100% means that the current OIs can handle all the workload. The second measurement Fig. 2(d)(e) is the accuracy which is calculated as the ratio between the current number of instances and the ideal one. An accuracy of 1 means that the actual number of nodes is the ideal one. An accuracy higher than 1 means that the operator has more instances than necessary. Finally, an accuracy of less than 1 means that we would need more instances if r is to be satisfied. An accuracy below one may delay tuples, but not necessarily as having $r < 1$ injects some safety.

Fig. 2(b)(c) shows that in window [0..50] and [100..150], the centralized approach outperforms the decentralized one but this is reversed in time window [50..100]: Even if the centralized method is more precise, global and deterministic, it is triggered only every 5 steps by the leader, where in the decentralized approach, replicas start scaling at different iterations. At each iteration potentially, a subset of the replicas start scaling. The same pattern can be seen which shows for each approach the ratio between the actual number of nodes and the ideal one. (See Fig. 2(d)(e)).

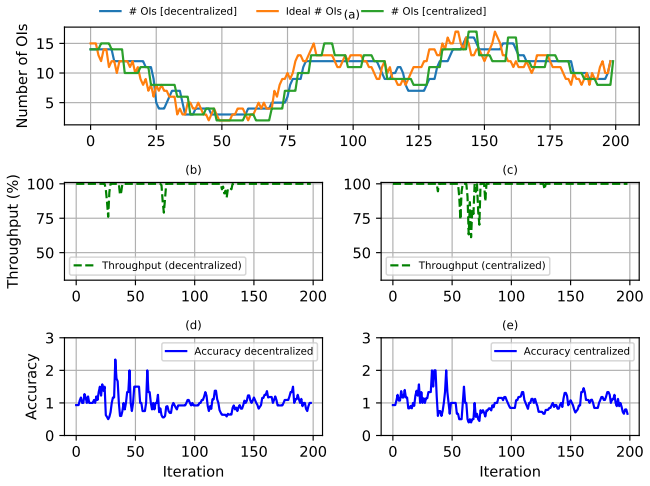

Fig. 2. Decentralized approach *vs* centralized approach

To estimate the added-value of the guarantee described in Section III-D, we used a different, less drastically changing workload, more precisely, sinusoid-based. The distance to the ideal number of nodes for one operator has been simulated both with and without the guarantee. Results are given in Figure 3. We can see that the guarantee is triggered notably in iterations 97-100 and 147-150, mitigating each time the amount of delayed tuples.

Finally, we estimated the overhead traffic due to our protocol when there are scaling operations. Let us first consider the traffic generated by a single duplication of one operator. Let us assume that according to the last variation in the load, k new nodes are needed, and that the current number of

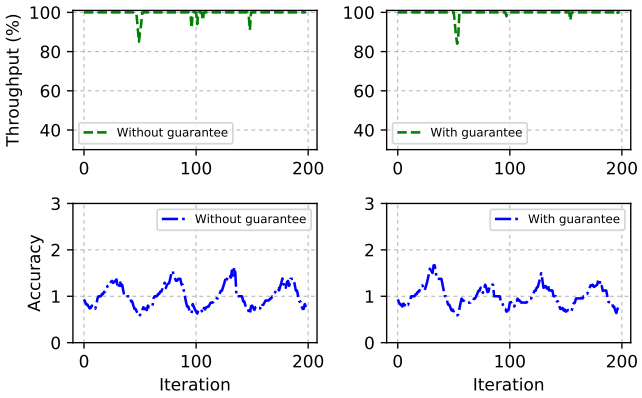

Fig. 3. Mitigating probability errors.

OI for the operator considered is n. Let us finally denote $succ$ and $pred$ the set of successors and predecessors for this operator, respectively. In the case of a single duplication conducted by one instance, the number of messages will be $2(|succs| + |preds|) + 1$: as detailed in Section III, the instance sends a duplication message to all its successors and predecessors, waits for their *acks*, and finally sends a start message to the new sibling. When k new siblings are to be created by the n current instances, there are two cases. If $k < n$, it means k instances trigger their duplication. If $k > n$, then all the instances trigger their duplication. Then the number of messages is then $2min(k, n)(|succs|+|preds|)+k$. Globally for the graph, the total traffic becomes quadratic in the number of nodes, as the previous result needs to be summed over all the levels, and the number of successors and predecessors appears then twice as a factor.

We conducted simulations to see the impact of overhead messages when the workload is artificially increased monotonically. Results are shown in Fig. 4.

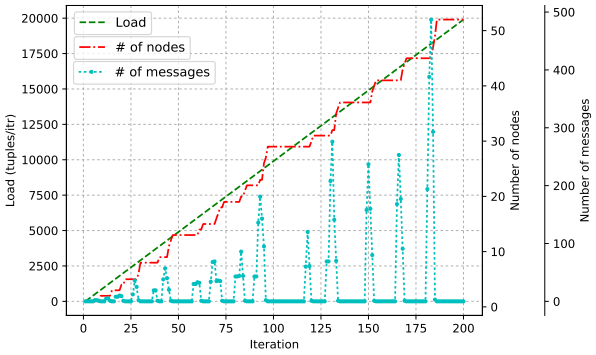

Fig. 4. Traffic when facing a monotonically increasing load.

The figure confirms an evolution of the overhead messages which is quadratic in the number of nodes (at iterations during which actual duplications are done), the number of nodes being, as already established, proportional to the workload.

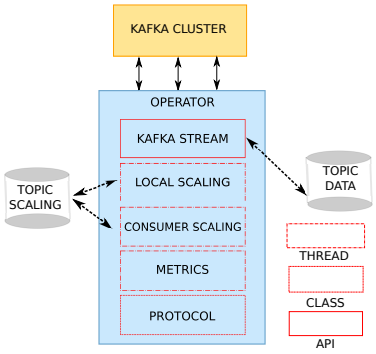

Fig. 5. Architecture of an operator's instance.

As stream processing applications can process hundreds of thousands data per second, the overhead messages sent on the network due to our protocol are in practice significantly lower than the workload of the data stream itself.

V. Towards a Decentralized Stream Engine

While simulation helped capturing our mechanism's behavior, an insight into its practical usage was missing. We started the development of a software prototype of a decentralized Stream Processing Engine, including the scaling mechanism described above. In this section, we describe its main characteristics and the technological choices made. We then present some experimental results obtained by deploying it over a computing cluster.

In the prototype, each instance becomes a process communicating with other processes, both for exchanging data streams and the control messages involved in the scaling protocol. The main technological choice made was to use Kafka and Kafka Streams [20]. Kafka, a high-level messaging middleware, is used to support the data streams between Kafka Streams operators. In other words, messaging between operators communicate using Kafka message queues (called *topics*) which are managed by the Kafka service. Kafka streams is provided as a library that can be used in any Java process that needs to become an operator in the pipeline. Due to this, the programmer has the ability to manipulate the Kafka Stream processes individually and build any scaling mechanism on top of it. Some other SPEs such as Storm includes more built-in mechanisms, which can be seen as a strength but this does not allow the user to easily implement its own scaling mechanism. Scaling a particular operator in Storm requires two steps, and relies on a centralized orchestrator: first, there is a need to start a new compute node and then ensure that the scheduler start this operator on this new compute node, which cannot be ensure natively without a serious modification of Storm's core. The architecture of an operator instance is shown in Fig. 5. When an instance gets started, three threads gets spawned: one is in charge of the scaling of the local operator, another one manages the messages the scaling protocol, and a third one collects metrics about the operator. Note that we use a single Kafka topic between contiguous operators in the pipeline. All instances of a given operator are

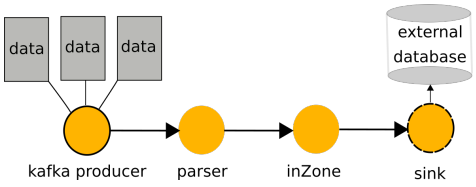

Fig. 6. Pipeline of the application.

part of the same Kafka *consumer group*: the topic's data are automatically dispatched amongst the members of the group. The scaling algorithm itself was implemented without any change. Each process is equipped with a local scaler triggerred periodically. Each process spawns new instances it decides. Protocol's messages are sent and received using Kafka as the communication middleware, using the particular Kafka topic (named *SCALING* in Fig. 5).

The prototype was deployed over Grid'5000 [3], a nation-wide platform gathering more than 8000 compute nodes. For the experiments, 12 Intel Xeon E5-2630 v3 with 8 cores each and 128 RAM interconnected by s 2x10 Gbps network. The actual spawning of the processes relied on a particular process making the interface with Grid'5000.

We used the dataset of the DEBS 2015 Grand Challenge [1], which gathers messages sent by Taxis in New York City over a period of twenty days (roughly 2 million events). Each record contains the information of a taxi's last trip (pick up time, drop off time, coordinates, duration) The application developed, illustrated in Fig. 6, filters trips which departs and stops in a specific subarea. The first operator acts as a data producer injecting the records into the pipeline. The second operator filters out invalid data. The third operator, called the `inZone` operator, keeps only trips within a specific area. The final operator is a simple sink counting trips. For the sake of stressing our prototype, the data stream was considerably accelerated: we made the load vary between 100 and 500 messages per second. The parameters of the experience was as follows: $thres_{\uparrow} = 0.8$, $thres_{\downarrow} = 0.6$, $r = 0.7$ and the period between two scaling operation was 10 sec. The results are shown in Figure 7. The middle (blue) curve shows the input rate and its variations: the input velocity was initially 200 messages sent per second. After 300 seconds, it is manually shifted to 500, before being reduced drastically after 600 seconds. The bottom curve shows the evolution of the number of instances triggered globally for the `inZone` operator. The top curve shows, for each replica started – note that processes appears and disappears with scaling operations –, its own send-rate, *i.e.*, the number of records it processed. The curves give a bit more confidence in the protocol's usability: the red curve mimics the blue one: even taken independently, scaling decisions allow to reach the required parallelism. Yet, the send-rate is not uniform amongst replicas, and also varies within one replica. In this experiment, we set the number of partitions for a Kafka topic to be 16. These 16 partitions are dispatched over the replicas. Such a distribution may

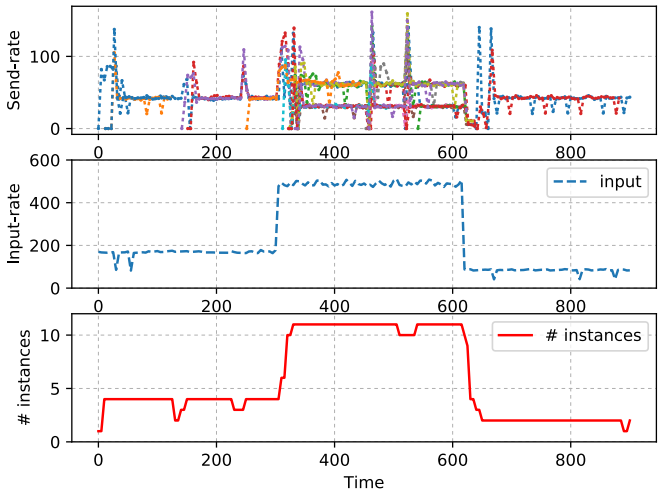

Fig. 7. Experimental results.

not be uniform, explaining the differences in the send-rate of replicas. Sometimes the send-rate drops suddenly to 0. These drops happen when scaling operations are triggered: when new instances appear in one Kafka consumer group, Kafka triggers a rebalancing phase to adapt the load balance amongst the updated set of replicas, affecting the throughput temporarily before a quick compensation. (Here, nodes are not fully-utilized as $r = 0.7$).

VI. Conclusion

This paper described few steps towards a decentralized stream processing engines, which is currently gaining momentum as geographically-distributed computing platforms are emerging. In particular, a fully-decentralized scaling mechanism was described, evaluated through simulations and implemented within a software prototype. The software prototype was deployed and evaluated over a real computing utility platform. This prototype constitutes an early stage of development paving the way for more complete framework for decentralized stream processing (D-SPE). Future work will consist in defining a proper API of a D-SPE and enhance it with deployment facility tools, such as container orchestrators or Cloud stacks. We also plan to further explore the stability of the protocol and propose mechanisms to avoid unnecessary fluctuations in the number of instances of operators.

Acknowledgment

This project was partially funded by ANR grant ASTRID SESAME ANR-16-ASTR-0026-02.

References

[1] The DEBS 2015 grand challenge. http://www.debs2015.org/call-grand-challenge.html.

[2] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive online scheduling in storm. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems (DEBS'13)*, pages 207–218, 2013.

[3] Daniel et al. Balouek. Adding virtualization capabilities to the Grid'5000 testbed. In *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer, 2013.

[4] Mehdi Belkhiria and Cédric Tedeschi. A Fully Decentralized Autoscaling Algorithm for Stream Processing Applications. In *Third International Workshop on Autonomic Solutions for Parallel and Distributed Data Stream Processing (Auto-DaSP 2019)*, Göttingen, Germany, August 2019. To appear. PDF available at https://hal.inria.fr/hal-02171172/file/autodasp2019.pdf.

[5] Nicolò M. Calcavecchia, Bogdan A. Caprarescu, Elisabetta Di Nitto, Daniel J. Dubois, and Dana Petcu. DEPAS: a Decentralized Probabilistic Algorithm for Auto-scaling. *Computing*, 94(8):701–730.

[6] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[7] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Distributed QoS-aware Scheduling in Storm. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS '15, pages 344–347, New York, NY, USA, 2015. ACM.

[8] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. Decentralized self-adaptation for elastic data stream processing. *Future Generation Comp. Syst.*, 87:171–185, 2018.

[9] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *ACM SIGMOD'13*, pages 725–736, 2013.

[10] Marcos Dias de Assunção, Alexandre Da Silva Veith, and Rajkumar Buyya. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. *J. Network and Computer Applications*, 103, 2018.

[11] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

[12] Ankit Toshniwal et al. Storm@twitter. In *International Conference on Management of Data (SIGMOD 2014)*, pages 147–156, Snowbird, USA, June 2014.

[13] Bugra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. Elastic scaling for data stream processing. *IEEE Trans. Parallel Distrib. Syst.*, 25(6), 2014.

[14] V. Gulisano, R. Jimnez-Peris, M. Patio-Martnez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365, Dec 2012.

[15] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Comput. Surv.*, 46(4), 2014.

[16] C. Hochreiner, M. Vgler, S. Schulte, and S. Dustdar. Elastic stream processing for the internet of things. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 100–107, June 2016.

[17] Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy Campbell. R-storm: Resource-aware scheduling in storm. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 149–161, 2015.

[18] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 49–49, April 2006.

[19] Scott Schneider, Martin Hirzel, Bugra Gedik, and Kun-Lung Wu. Auto-parallelizing Stateful Distributed Streaming Applications. In *International Conference on Parallel Architectures and Compilation Techniques, PACT '12*, pages 53–64, Minneapolis, USA, September 2012.

[20] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. Building a replicated logging system with apache kafka. *PVLDB*, 8(12):1654–1655, 2015.

[21] J. Xu, Z. Chen, J. Tang, and S. Su. T-storm: Traffic-aware online scheduling in storm. In *IEEE 34th International Conference on Distributed Computing Systems*, 2014.

[22] Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P. Jue. All One Needs to Know about Fog Computing and Related Edge Computing Paradigms: A Complete Survey. *CoRR*, abs/1808.05283, 2018.

[23] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: fault-tolerant streaming computation at scale. In *24th ACM SIGOPS Symposium on Operating Systems Principles (SOSP'13)*, pages 423–438, Farmington, USA, November 2013.